CS 839: Design the Next-Generation Database
Lecture 19: RDMA for OLAP

Xiangyao Yu
3/31/2020
Discussion Highlights

SmartNIC vs. SmartSSD
- Different application scenarios: one for storage, one for network
- SATA vs. PCIe?
- SmartNICs used for reducing CPU overhead; SmartSSD used for reducing data movement
- SmartNIC seems more popular among hardware vendors
- Computation in SmartNIC is stronger than SmartSSD

Database operators pushed to SmartNIC
- Common: encryption, caching
- OLTP: filtering, aggregation, locking, indexing
- OLAP: filtering, project, aggregation, compression

Benefits of putting smartness into the NIC
- Packet processing, latency reduction
- Effect of SmartSSD is limited due to caching; caching does not apply in SmartNIC
- Isolate security checks from CPU
- Collect run time statistics such as network usage and latencies
- Reduces burden on PCIe
Distributed Join Algorithms on Thousands of Cores

Claude Barthels, Ingo Müller†, Timo Schneider, Gustavo Alonso, Torsten Hoefler
Systems Group, Department of Computer Science, ETH Zurich
{firstname.lastname}@inf.ethz.ch

ABSTRACT
Traditional database operators such as joins are relevant not only in the context of database engines but also as a building block in many computational and machine learning algorithms. With the advent of big data, there is an increasing demand for efficient join algorithms that can scale with the input data size and the available hardware resources.

In this paper, we explore the implementation of distributed join algorithms in systems with several thousand cores connected by a low-latency network as used in high performance computing systems or data centers. We compare radix hash join to sort-merge join algorithms and discuss their implemen-

This paper addresses the challenges of running state-of-the-art, distributed radix hash and sort-merge join algorithms at scales usually reserved to massively parallel scientific applications or large map-reduce batch jobs. In the experimental evaluation, we provide a performance analysis of the distributed joins running on 4,096 processor cores with up to 4.8 terabytes of input data. We explore how join algorithms behave when high-bandwidth, low-latency networks are used and specialized communication libraries replace hand-tuned code. These two points are crucial to understand the evolution of distributed joins and to facilitate the portability of the implementation to future systems.
Bandwidth and Latency

![Bandwidth and Latency Chart]

- **10 Gbit Ethernet**
- **SSD**
- **RDMA Network (HDR 4x links)**
- **NVDIMM**
- **Remote DRAM**
- **Local DRAM**
- **LLC cache**
- **L1 cache**
# Algorithm Designs

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory</th>
<th>RDMA &amp; SmartNIC</th>
<th>Distributed System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency Control</td>
<td>Shared lock table</td>
<td>???</td>
<td>Partitioned lock table</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Shared log</td>
<td>???</td>
<td>Two-phase commit</td>
</tr>
<tr>
<td>Join</td>
<td>Radix join</td>
<td>???</td>
<td>Bloom-filter + semi-join</td>
</tr>
</tbody>
</table>
Message Passing

Shared memory

Message Passing
Message Passing Interface (MPI)

Standard library interface for writing parallel programs in high-performance computing (HPC)

• Hardware independent interface
• Can leverage performance of underlying hardware
**Memory Window:** memory that is accessible by other processes through RMA operations
MPI One-Sided Operations

**MPIWin_create**: exposes local memory to RMA operation by other processes.
- Collective operation
- Creates window object

**MPIWin_free**: deallocates window object

**MPIPut**: moves data from local memory to remote memory

**MPIGet**: retrieves data from remote memory into local memory

**MPIWin_lock** and **MPIWin_unlock** to protect RMA operations on a specific window
Radix Hash Join

Partitioned hash join achieves the best performance when each partition of the inner relation fits in cache

⇒ A large number of partitions

⇒ Performance suffers when the # partitions > # TLB entries or # of cachelines in the cache

Radix Join: Partition through multiple passes
Radix Hash Join

1\textsuperscript{st} pass of partitioning
Radix Hash Join

1st pass of partitioning

Data shuffle
Radix Hash Join

1\textsuperscript{st} pass of partitioning

Data shuffle

Following passes of partitioning
Radix Hash Join

1\textsuperscript{st} pass of partitioning

Data shuffle

Following passes of partitioning

Partition outer relation
Radix Hash Join

1st pass of partitioning
Data shuffle
Following passes of partitioning
Build and probe
Partition outer relation
Radix Hash Join – Performance Model

Compute the histogram
• Determine the size of memory windows
• Assignment of partitions to nodes
• Offsets within memory windows into which each process writes exclusively

\[ T_{\text{hist}} = \frac{|R| + |S|}{p \cdot P_{\text{scan}}} \]
Radix Hash Join – Performance Model

Multi-pass partitioning

Number of passes

\[ d = \lceil \log_{F_P} (|R| / \text{cache size}) \rceil \]

\( F_P \) : partitioning fan-out

Time of partitioning

\[ T_{\text{part}} = \left( \frac{1}{p \cdot P_{\text{net}}} + \frac{d - 1}{p \cdot P_{\text{part}}} \right) \cdot (|R| + |S|) \]

\[ P_{\text{net}} = \min \left( P_{\text{part}}, \frac{BW_{\text{node}}}{t} \right) \]
Radix Hash Join – Performance Model

Build and Probe

Build Time

\[ T_{\text{build}} = (F_P)^d \cdot \frac{|R_p|}{p \cdot P_{\text{build}}} = \frac{|R|}{p \cdot P_{\text{build}}} \]

Probe Time

\[ T_{\text{probe}} = (F_P)^d \cdot \frac{|S_p|}{p \cdot P_{\text{probe}}} = \frac{|S|}{p \cdot P_{\text{probe}}} \]
Radix Hash Join – Performance Model

\[ T_{\text{rdx}} = T_{\text{hist}} + T_{\text{part}} + T_{\text{build}} + T_{\text{probe}} \]

\[ = \frac{|R| + |S|}{p \cdot P_{\text{scan}}} \]

\[ + \left( \frac{1}{p \cdot P_{\text{net}}} + \frac{d - 1}{p \cdot P_{\text{part}}} \right) \cdot (|R| + |S|) \]

\[ + \frac{|R|}{p \cdot P_{\text{build}}} \]

\[ + \frac{|S|}{p \cdot P_{\text{probe}}} \]
Sort-Merge Join

Range partitioning
Sort-Merge Join

Range partitioning
Sort individual runs
Sort-Merge Join

Range partitioning
Sort individual runs
Data shuffle
Sort-Merge Join

Range partitioning
Sort individual runs
Data shuffle
Merge
Sort-Merge Join

Range partitioning
Sort individual runs
Data shuffle
Merge
Sort-merge outer relation
Sort-Merge Join

Range partitioning
Sort individual runs
Data shuffle
Merge
Join
Sort-merge outer relation
Sort-Merge Join – Performance Model

Partitioning

\[ T_{part} = \frac{|R| + |S|}{p \cdot P_{part}} \]
Sort-Merge Join – Performance Model

Sorting individual runs of length \( l \)

**Number of runs**

\[
N_R = \frac{|R|}{l} \quad \text{and} \quad N_S = \frac{|S|}{l}
\]

**Sorting performance**

\[
P_{\text{sort}} = \min \left( P_{\text{run}}(l), \frac{BW_{\text{node}}}{t} \right)
\]

**Sorting time**

\[
T_{\text{sort}} = (N_R + N_S) \cdot \frac{l}{p \cdot P_{\text{sort}}} = \frac{|R| + |S|}{p \cdot P_{\text{sort}}}
\]
Sort-Merge Join – Performance Model

Merging multiple runs into a sorted output

Number of iterations
\[ d_R = \left\lfloor \log_{F_M} \left( \frac{N_R}{p} \right) \right\rfloor \quad \text{and} \quad d_S = \left\lfloor \log_{F_M} \left( \frac{N_S}{p} \right) \right\rfloor \]

\( F_M \): Merge fan-in

Merge time
\[ T_{\text{merge}} = d_R \cdot \frac{|R|}{p \cdot P_{\text{merge}}} + d_S \cdot \frac{|S|}{p \cdot P_{\text{merge}}} \]
Sort-Merge Join – Performance Model

Joining sorted relations

\[ T_{\text{match}} = \frac{|R| + |S|}{p \cdot P_{\text{scan}}} \]
Sort-Merge Join – Performance Model

Total execution time

\[ T_{sm} = T_{part} + T_{sort} + T_{merge} + T_{match} \]

\[ = \frac{|R| + |S|}{p \cdot P_{part}} + \frac{|R| + |S|}{p \cdot P_{sort}} + d_R \cdot \frac{|R|}{p \cdot P_{merge}} + d_S \cdot \frac{|S|}{p \cdot P_{merge}} + \frac{|R| + |S|}{p \cdot P_{scan}} \]
Radix-Hash Join vs. Sort-Merge Join

Radix join

\[ T_{rdx} = T_{hist} + T_{part} + T_{build} + T_{probe} \]

\[ = \frac{|R| + |S|}{p \cdot P_{scan}} \]

\[ + \left( \frac{1}{p \cdot P_{net}} + \frac{d - 1}{p \cdot P_{part}} \right) \cdot (|R| + |S|) \]

\[ + \frac{|R|}{p \cdot P_{build}} \]

\[ + \frac{|S|}{p \cdot P_{probe}} \]

Sort-merge join

\[ T_{sm} = T_{part} + T_{sort} + T_{merge} + T_{match} \]

\[ = \frac{|R| + |S|}{p \cdot P_{part}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{sort}} \]

\[ + d_R \cdot \frac{|R|}{p \cdot P_{merge}} + d_S \cdot \frac{|S|}{p \cdot P_{merge}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{scan}} \]
Radix-Hash Join vs. Sort-Merge Join

Radix join

\[ T_{\text{rdx}} = T_{\text{hist}} + T_{\text{part}} + T_{\text{build}} + T_{\text{probe}} = \frac{|R| + |S|}{p \cdot P_{\text{scan}}} \]

\[ + \left( \frac{1}{p \cdot P_{\text{net}}} + \frac{d - 1}{p \cdot P_{\text{part}}} \right) \cdot (|R| + |S|) \]

\[ + \frac{|R|}{p \cdot P_{\text{build}}} \]

\[ + \frac{|S|}{p \cdot P_{\text{probe}}} \]

Sort-merge join

\[ T_{\text{sm}} = T_{\text{part}} + T_{\text{sort}} + T_{\text{merge}} + T_{\text{match}} = \frac{|R| + |S|}{p \cdot P_{\text{part}}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{\text{sort}}} \]

\[ + d_R \cdot \frac{|R|}{p \cdot P_{\text{merge}}} + d_S \cdot \frac{|S|}{p \cdot P_{\text{merge}}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{\text{scan}}} \]
Radix-Hash Join vs. Sort-Merge Join

Radix join

\[ T_{rdx} = T_{hist} + T_{part} + T_{build} + T_{probe} \]

\[ = \frac{|R| + |S|}{p \cdot P_{scan}} \]

\[ + \left( \frac{1}{p \cdot P_{net}} + \frac{d-1}{p \cdot P_{part}} \right) \cdot (|R| + |S|) \]

\[ + \frac{|R|}{p \cdot P_{build}} \]

\[ + \frac{|S|}{p \cdot P_{probe}} \]

Sort-merge join

\[ T_{sm} = T_{part} + T_{sort} + T_{merge} + T_{match} \]

\[ = \frac{|R| + |S|}{p \cdot P_{part}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{sort}} \]

\[ + d_R \cdot \frac{|R|}{p \cdot P_{merge}} + d_S \cdot \frac{|S|}{p \cdot P_{merge}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{scan}} \]
Radix-Hash Join vs. Sort-Merge Join

Radix join

$$T_{rdx} = T_{hist} + T_{part} + T_{build} + T_{probe}$$

$$= \frac{|R| + |S|}{p \cdot P_{scan}} + \left( \frac{1}{p \cdot P_{net}} + \frac{d - 1}{p \cdot P_{part}} \right) \cdot (|R| + |S|)$$

$$+ \frac{|R|}{p \cdot P_{build}} + \frac{|S|}{p \cdot P_{probe}}$$

Sort-merge join

$$T_{sm} = T_{part} + T_{sort} + T_{merge} + T_{match}$$

$$= \frac{|R| + |S|}{p \cdot P_{part}}$$

$$+ \frac{|R| + |S|}{p \cdot P_{sort}}$$

$$+ d_R \cdot \frac{|R|}{p \cdot P_{merge}} + d_S \cdot \frac{|S|}{p \cdot P_{merge}}$$

$$+ \frac{|R| + |S|}{p \cdot P_{scan}}$$
Radix-Hash Join vs. Sort-Merge Join

Radix join

\[ T_{rdx} = T_{hist} + T_{part} + T_{build} + T_{probe} \]

\[ = \frac{|R| + |S|}{p \cdot P_{scan}} + \left( \frac{1}{p \cdot P_{net}} + \frac{d-1}{p \cdot P_{part}} \right) \cdot (|R| + |S|) \]

\[ + \frac{|R|}{p \cdot P_{build}} + \frac{|S|}{p \cdot P_{probe}} \]

Sort-merge join

\[ T_{sm} = T_{part} + T_{sort} + T_{merge} + T_{match} \]

\[ = \frac{|R| + |S|}{p \cdot P_{part}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{sort}} \]

\[ + d_R \cdot \frac{|R|}{p \cdot P_{merge}} + d_S \cdot \frac{|S|}{p \cdot P_{merge}} \]

\[ + \frac{|R| + |S|}{p \cdot P_{scan}} \]
Performance Evaluation
Baseline Experiments

Radix hash join on Cray XC30 (MPI)
without data compression

Radix hash join on rack-scale system (InfiniBand)
without data compression

Extrapolated performance
Scale-Out Experiments

- Compression improves performance
- Radix join outperforms sort-merge join

![Graph showing throughput vs number of cores for different join methods with and without data compression.](image-url)
Radix Join Execution Time Breakdown

Time of Histogram computation and window allocation largely remains constant
Radix Join Execution Time Breakdown

Time of local partitioning and build/probe remain constant
Radix Join Execution Time Breakdown

Time of network partitioning increases at more than 1024 cores
Radix Join Execution Time Breakdown

Time of network partitioning increases at more than 1024 cores
- Partitioning fan-out is increased beyond its optimal setting
- Additional time spent in MPI_Put and MPI_Flush
Radix Join Execution Time Breakdown

Time due to load imbalance increases with core count
Sort-Merge Join Execution Time Breakdown

(a) Total join execution

(b) Sorting phase
Partitioning fan-out is pushed beyond its optimal configuration
Sort-Merge Join Execution Time Breakdown

Within sorting, time of network shuffling increases with core count
Sort-Merge Join Execution Time Breakdown

Time of merge and joining stays constant
Time due to load imbalance slightly increases with core count
Scale-Up Experiments

With more cores per machine, considerably more time spent on `MPI_Put` and `MPI_Flush`. Difficult to fully interleave computation and communication.
Comparison with the Model

<table>
<thead>
<tr>
<th>Radix hash join</th>
<th>Sort-merge join</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase</strong></td>
<td><strong>Partitioning</strong></td>
</tr>
<tr>
<td>Histogram Comp.</td>
<td>1.20s</td>
</tr>
<tr>
<td>Window Allocation</td>
<td>0.06s</td>
</tr>
<tr>
<td><strong>Network Partitioning</strong></td>
<td>2.08s</td>
</tr>
<tr>
<td>Local Partitioning</td>
<td>0.58s</td>
</tr>
<tr>
<td>Build-Probe</td>
<td>0.51s</td>
</tr>
<tr>
<td>Imbalance</td>
<td>0.62s</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>5.70s</td>
</tr>
<tr>
<td><strong>Exec. Time</strong></td>
<td>5.70s</td>
</tr>
<tr>
<td><strong>Model</strong></td>
<td>4.61s</td>
</tr>
<tr>
<td><strong>Diff.</strong></td>
<td>+1.09s</td>
</tr>
</tbody>
</table>

**Parameters** [million tuples per second]

 RHJ: $P_{\text{scan}} = 225$, $P_{\text{part}} = 120$, $P_{\text{net}} = 1024$, $P_{\text{build}} = 120$, $P_{\text{probe}} = 225$

 SMJ: $P_{\text{part}} = 78$, $P_{\text{sort}} = 75$, $P_{\text{net}} = 1024$, $P_{\text{merge}} = 45$, $P_{\text{scan}} = 225$

Network shuffling is the bottleneck
RDMA for OLAP – Q/A

Collective communication scheduling for joins?

Supercomputers used in the real world for database workloads?

Radix join vs. hash join?

Radix join does not achieve theoretical maximum performance

What is partition fan-out?

MPI vs. shared memory for join
Group Discussion

How can Smart NICs help improve the performance of joins?

Can you think of any hardware/software techniques that may close the performance gap between radix join and sort-merge join?

Can you think of any hardware/software techniques that may allow radix join to achieve its theoretical maximum performance?
Before Next Lecture

Submit discussion summary to https://wisc-cs839-ngdb20.hotcrp.com
  • Deadline: Wednesday 11:59pm

Submit review for
  • Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases
  • [optional] Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes